AFRL-VA-WP-TR-2005-3029

ASPECT SUITE AUTOMATION FOR EMBEDDED MISSION SYSTEMS

Brian J. Ellis

The Boeing Company
P.O. Box 516
St. Louis, MO 63166

John A. Stankovic

University of Virginia

MARCH 2005

Final Report for 28 July 2000 – 31 January 2005

Approved for public release; distribution is unlimited.

STINFO FINAL REPORT

AIR VEHICLES DIRECTORATE
AIR FORCE MATERIEL COMMAND
AIR FORCE RESEARCH LABORATORY
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7542

NOTICE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the Air Force Research Laboratory Wright Site Public Affairs Office (AFRL/WS) and is releasable to the National Technical Information Service (NTIS). It will be available to the general public, including foreign nationals.

THIS TECHNICAL REPORT IS APPROVED FOR PUBLICATION.

/s/

DANIEL J. SCHREITER
Program Manager

/s/

MICHAEL P. CAMDEN
Chief, Control Systems Development
and Applications Branch
Air Force Research Laboratory

This report is published in the interest of scientific and technical information exchange and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
March 2005	Final	07/28/2000 – 01/31/2005

4. TITLE AND SUBTITLE
ASPECT SUITE AUTOMATION FOR EMBEDDED MISSION SYSTEMS

5a. CONTRACT NUMBER
F33615-00-C-3048

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
0602302

6. AUTHOR(S)
Brian J. Ellis (The Boeing Company)
John A. Stankovic (University of Virginia)

5d. PROJECT NUMBER
A04I

5e. TASK NUMBER

5f. WORK UNIT NUMBER
0A

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

The Boeing Company
P.O. Box 516
St. Louis, MO 63166

University of Virginia

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Vehicles Directorate
Air Force Research Laboratory
Air Force Materiel Command
Wright-Patterson Air Force Base, OH 45433-7542

DARPA

10. SPONSORING/MONITORING AGENCY ACRONYM(S)
AFRL/VACC

11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)
AFRL-VA-WP-TR-2005-3029

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
Report contains color.

14. ABSTRACT

Aspect oriented programming (AOP), when used well, has many advantages. Aspects are however, programming-time constructs, i.e., they relate to source code. In this project, two types of design time aspects were identified, aspect checks and prescriptive aspects and these concepts were incorporated into a compositional toolkit; VEST. The VEST toolkit can substantially improve the development, implementation and evaluation of systems built from components which must interoperate, satisfy various dependencies, and meet non-functional requirements. The toolkit focuses on using language independent notions of aspects to deal with distributed embedded system issues that include application domain specific code, middleware, the OS, prescriptive aspects, and the hardware platform.

15. SUBJECT TERMS
Distributed embedded systems, aspects, prescriptive aspects, software composition tools, software analysis tools, component based design

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT:	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON (Monitor)
a. REPORT	b. ABSTRACT	c. THIS PAGE			Daniel J. Schreiter
Unclassified	Unclassified	Unclassified	SAR	46	19b. TELEPHONE NUMBER *(Include Area Code)* (937) 255-8291

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39-18

TABLE OF CONTENTS

1. Introduction

This document is the final report for the VEST research project. VEST is funded by the DARPA PCES program under grant F33615-00-C-3048.

Success of distributed embedded systems depends on low cost, quickness to market, and in some cases, flexible operation of the product. The reliability of these products and the degree of configurability are paramount concerns, and, in many cases, there are important real-time constraints that have to be met. Building distributed embedded system software is time-consuming and costly. The use of software components for constructing and tailoring these systems has promise. What are needed are tools to support program *composition* and *analysis* of component-based embedded systems. In these systems designs are instantiated largely by choosing pre-written components from libraries rather than by implementing the design from scratch. Composition tools are different from top-down design tools (e.g., Rational Rose) that do not directly support composition of pre-existing components. One major difficulty of embedded system composition is the crosscutting dependencies among components that are often hidden from the composers. Composition tools should support dependency checks across component boundaries and expose potential composition errors due to the crosscutting dependencies.

Our work focuses on the development of effective composition mechanisms, and the associated dependency and nonfunctional analyses for real-time embedded systems. Our solution is based on extending the notion of *aspects*. Aspects are defined as those issues that cannot be cleanly encapsulated in a generalized procedure because they are characterized as being systemic. They usually include issues that affect the performance or semantics of components. This includes many real-time, concurrency, synchronization, and reliability issues. Aspects, to date, have largely been language dependent in that aspects are implemented as language constructs. A major contribution of our work is that we extend the concept of aspects to language independent notions and apply them at design time. Our solutions are embodied within a toolkit called VEST (Virginia Embedded Systems Toolkit). VEST provides an environment for the composition and analysis of distributed real-time embedded systems. VEST models application components, middleware, OS, and hardware components. This feature supports the composition and tailoring of every layer in an embedded system for a specific application, which leads to more complete crosscutting dependency checks and more optimization opportunities. VEST itself is not a complete requirements, design and implementation tool; rather it currently focuses on the specific composition and analysis tasks.

VEST includes features that are found in other tools. However, there are several novel features in VEST. The major contributions of VEST are two types of language-independent aspects referred to as *aspect checks* and *prescriptive aspects*.

Together these permit the benefits of aspects to be exercised early in the composition process rather than in the implementation phase. A set of representative aspect checks in embedded software is identified and implemented in VEST. Some of these aspects are simple dependency checks; others are complex and may involve the entire system, e.g., distributed real-time scheduling. The simple fact of identifying key aspect checks improves our understanding of specific crosscutting concerns found in distributed embedded systems, including middleware. Prescriptive aspects allow application specific advice to be applied to designs and they have a global effect. The significance of VEST is largely derived from language-independent aspects.

VEST has been integrated with other tools in the PCES and MoBIES projects. An interface was built that allows VEST to invoke a commercial real-time analysis tool called TimeWiz. This provides detailed rate monotonic analysis for many types of real-time systems. VEST also produces XML output so that the Boeing configuration tools can be used to construct the executables. VEST also includes a mapping tool that converts Boeing Boldstroke middleware Avionics Component Library (ACL) components to VEST-readable components. The Implementation Interface Format (IIF) from Boeing was mapped to VEST, thereby enabling some automatic assignment of timing attributes for components.

This report is structured as follows: section 2 presents a high level overview of VEST. Section 3 describes various aspect checks, including schedulability check, memory footprint check, event dependency check and buffer size check. Section 4 discusses prescriptive aspects. Section 5 gives more details on the Virginia Prescriptive Aspect Language (VPAL). Section 6 uses 3 case studies to illustrate the functionalities of VEST, including experiments that measure the amount of productivity gains from using VEST. Section 7 summarizes the versions of VEST that have been delivered. Section 8 provides a short summary. Appendix A is the BNF grammar definition of VPAL. BNF is an acronym for "Backus Naur Form" - a formal notation to describe the syntax of a given language.

2. Overview of VEST

VEST provides an environment for constructing and analyzing component-based distributed real-time embedded systems. VEST helps developers select or create passive software components, compose them into a product, map the passive components onto active structures such as threads, map threads onto specific hardware, and perform dependency checks and non-functional analyses to offer as many guarantees as possible along many dimensions including real-time performance and reliability. Distributed embedded systems issues are explicitly addressed via the mapping of components to active threads and to hardware, the ability to include middleware as components, and the specification of a network and distributed nodes.

The VEST environment is composed of five libraries, a set of aspect checks, and a GUI-based environment for composing and analyzing embedded products.

- **Component Libraries**: Because VEST supports real-time distributed embedded systems, the VEST component libraries contain both software and descriptions of hardware components and networks. VEST components can be *abstract* or *actual*. An abstract component is a design entity that represents the requirements, e.g., a timer with certain requirements or a generic processor is an abstract component. An actual component is the implementation or description of a reusable entity. A specific timer module written in C and a Motorola MPC7455 are examples of actual components. Sets of reflective information exist for each of these component types. The reflective information of an abstract component includes its interface and requirements such as for security. The reflective information for actual components includes categories such as linking information, location of source code, worst-case execution time, memory footprint, and other reflective information needed to analyze crosscutting dependencies. The extent of the reflective information and its extensibility are some of the key features that distinguish VEST from many other tools (see section 6). To support the whole design process of embedded systems, VEST implements the following four component libraries each for a separate software/hardware layer:
- **Application Library** includes software components for a particular application domain. For example, an avionics application library includes a set of navigation, planning, sensor fusion, and pilot display components. Currently, application components in VEST are CORBA components.
- **Middleware Library** includes components of the middleware. For example, a Real-Time CORBA library includes different CORBA service modules such as scheduling services and persistence services.

- **OS Library** includes components of operating systems. For example, threads are OS components in VEST and also have reflective information describing their attributes such as invocation period and scheduling priorities.
- **Hardware Library** includes descriptions of hardware components such as processors, RAM, NVRAM, buses, network connections, DSP, A/D and D/A, actuators and sensors.
- **Prescriptive Aspects Library:** Prescriptive aspects are reusable programming language independent advice that may be applied to a design. For example, a developer can invoke a set of prescriptive aspects in the library to add a certain security mechanism *en masse* to an avionics product.
- **Aspect Checks:** VEST implements both a set of simple intra- and inter-component aspect checks that crosscut component boundaries. A developer can apply these checks to a system design to discover errors caused by dependencies among components. One aspect check in VEST is the real-time schedulability analysis for both single-node and distributed embedded systems. VEST can also invoke off-the-shelf analysis tools from its GUI environment.
- **Composition Environment:** VEST provides a GUI-based environment that lets developers compose distributed embedded systems from components, perform dependency checks, and invoke prescriptive aspects on a design. As shown in figure 1 below, the GUI of VEST displays four main panels. The main canvas contains the product under development. At first, this contains abstract components that describe the design. The user then chooses actual components from libraries to instantiate the design. Actual components also appear on this main canvas. The second graphical panel (on the right hand side) displays the structure of the product under development. The third panel (on the bottom left) displays all the components in a particular component library once it is chosen. The fourth panel (on the lower right) displays all the attributes (reflective information) of a particular component when that component is highlighted. The developer can invoke an aspect check by clicking on a corresponding button on the menu bar. He can also apply a prescriptive aspect by invoking an aspect interpreter from a button on the menu bar.

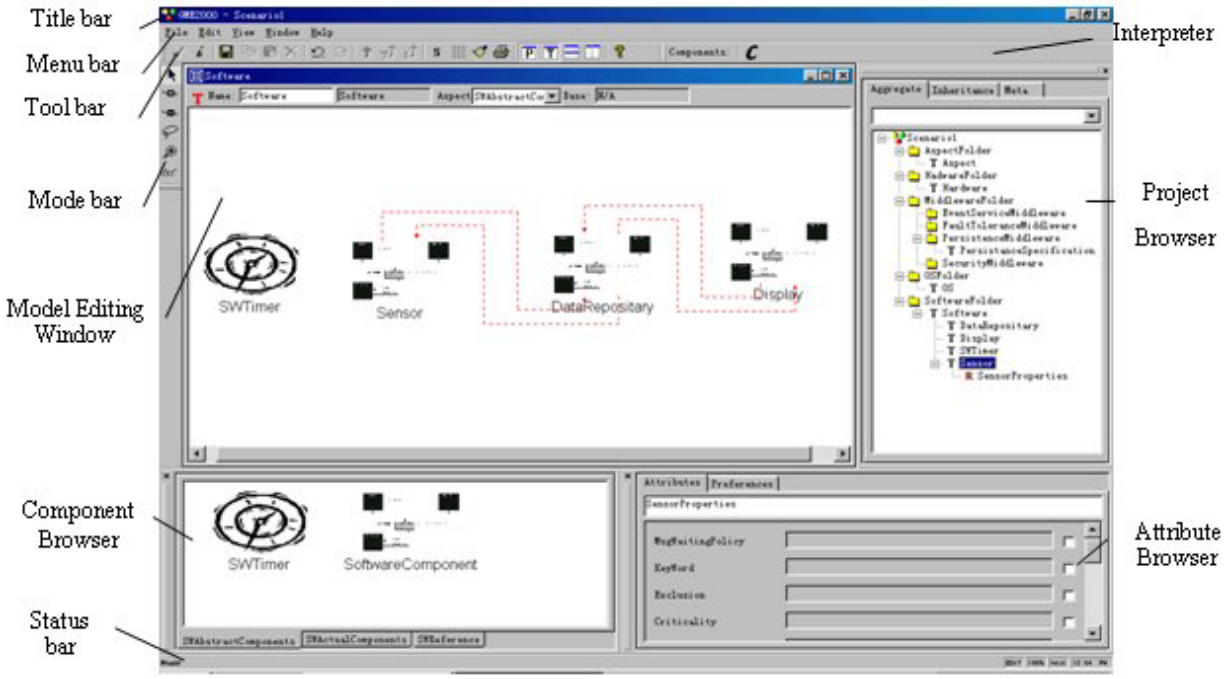


Figure 1: VEST Composition Environment

From the VEST GUI, a system developer can compose a distributed embedded system in the following way:

1) Design a product by choosing and combining abstract components from the libraries. In the future, designs could be imported from a requirements tool , e.g., from Rose models based on UML. To date, we have implemented a mapping program that permits component descriptions based on UML to be imported into VEST libraries.
2) Design the distributed systems hardware platform by choosing and combining abstract components from the libraries.
3) Map software components to hardware and threads so that the active part of a composed system can be designed and analyzed. Only after this step can we truly do the real-time analysis since execution times are highly platform dependent.
4) Synthesize the product by instantiating abstract components with actual components. It is possible to create a hierarchy of components.
5) Apply prescriptive aspects. This is one area where VEST makes a major contribution. Previous systems do not have enough support for crosscutting dependencies among components and this is one advantage of VEST.
6) Perform aspect checks and invoke (internal and off-the-shelf) analysis tools to analyze a configured system. If some checks fail, the developer may need to reconfigure or replace the actual components and repeat the checks.

VEST also provides a separate GUI for system administrators to maintain the libraries

and checks. From this interface, a system administrator can create a new abstract or actual component. Specifying components entails supplying a significant amount of validated reflective information. He can also add or delete prescriptive aspects and dependency checks.

3. Aspect Checks

One goal of VEST is to provide support for various types of dependency checking among components during the composition process. Dependency checks are invoked to establish certain properties of the composed system. This is a critical part of real-time embedded system design and implementation. Some dependency checks are simple and have been understood for a long time. We call these intra- and inter-component dependency checks. Other dependencies are very difficult and even insidious. We refer to these as crosscutting dependencies or *aspect checks*. Aspect checking is an explicit check across components that exist in the current product configuration. We have identified many aspect checks that would help a developer avoid difficult to find errors when creating embedded systems from components. In many cases the important thing is identifying the check required and implementing it so that it is automatic. Although the implementation of some checks may be simple, when these checks are combined with all the other features of VEST, the result is a powerful tool.

Aspect checks verify certain properties of a real-time embedded system design. Aspect checks are explicit checks across components in a system. Usually an aspect check looks for hidden dependencies among components that are hard to directly identify by a designer. There are various kinds of “global” hidden dependencies in a system design. We focus on the most interesting checks to designers in this avionics application. In the domain of avionics systems, our aspect checks include a memory footprint check, an event channel check, a buffer size check, and schedulability analysis.

Developers can invoke checks on the current product from the GUI environment. In general, it is our belief that aspects (both aspect checks and prescriptive aspects) include an open ended set of issues. Therefore, we cannot hope to be complete. Rather we need to identify key aspects for embedded systems and create the specification and tools to address as many of these issues as possible. The more aspect checks that can be performed, the more confidence in the resulting composed system we will have. However, by no means do we claim that the system is correct; only that certain specific checked errors are not present.

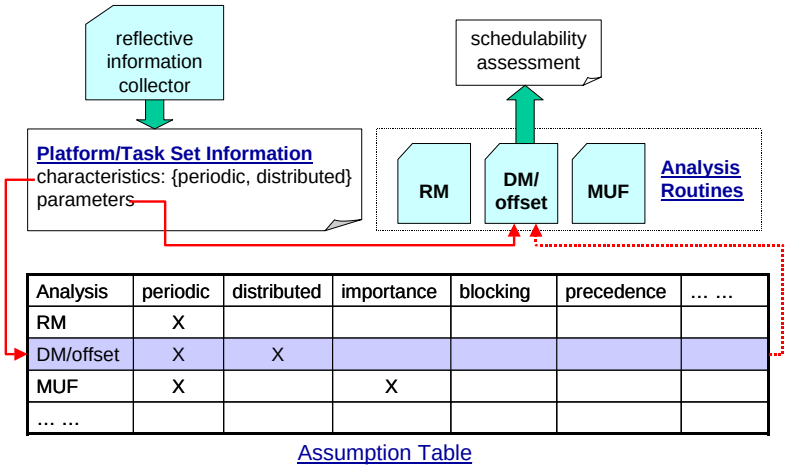


Figure 2: Schedulability analysis in VEST

3.1 Real-Time Schedulability Check

An important check for real-time embedded systems is the schedulability analysis that validates whether all tasks can make their deadlines. Note that while designing and implementing a system that most changes made will affect the real-time properties of the system. This makes real-time scheduling a global cross cutting dependency. While many different schedulability analysis techniques exist, they differ in their assumptions on the task set and none of the existing analysis is applicable to all real-time embedded systems. The compatibility between schedulability analyses and the characteristics of the designed system is a typical crosscutting dependency that is "hidden" from the designer. Using an incompatible analysis on a system can lead to timing violations even when the schedulability analysis itself is correct. To handle different types of embedded systems, VEST provides a flexible and extensible scheduling tool that provides aspect checks on the compatibility between existing schedulability analyses and the system. This tool (shown in figure 2 above) is composed of a set of schedulability analysis routines, an assumption table, and a reflective information collector. The assumption table lists the assumptions of each schedulability analysis routine. The current list of assumptions includes:

Periodic: are all the tasks periodic?

Distributed: are any of the tasks distributed on multiple processors?

Importance: are important tasks protected in overload conditions?

Blocking: can low priority tasks block high priority tasks?

Precedence: are there precedence constraints among tasks?

For example, the assumptions of the Rate Monotonic analysis are that all tasks are periodic. The Rate Monotonic with Priority Ceiling protocol's assumptions are (periodic, blocking). The VEST scheduling tool is extensible and new scheduling techniques can be added to the tool together with their assumptions.

Developers can assess the schedulability of the current design by running the scheduling tool from the GUI. The reflective information collector scans the software, hardware and network components of the design and produces a platform/task set information file that includes a list of the characteristics and the timing information of the task set. The tool selects an analysis whose assumptions match the characteristics of the system. This ensures that proper analysis and scheduling policy is applied. For example, for a system with all independent periodic tasks on a single processor, the Rate Monotonic Analysis (RMA) check or Maximum Urgency First (MUF) will be applied to the system. However, if the same task set is designed on a distributed platform, the DM/Offset analysis described below will be applied.

3.1.1 Deadline Monotonic Scheduling with Phase Offset

Currently the VEST scheduling tool implements the basic Rate Monotonic check, the Maximum Urgency First algorithm, and a more sophisticated end-to-end analysis for distributed systems. In applying the tool to a Boeing's distributed avionics case study, we found that RMA and MUF were not sufficient because such systems often run on a distributed platform. Avionics based on real-time CORBA (e.g., Bold Stroke and TAO) requires support for the following distributed scheduling problem (see figure3 below).

A periodic task T_i consists of multiple subtasks $\{T_{ij}\}$ on different processors. The set of subtasks have the same period P_i, and the task deadline $D_i = P_i$. Figure 3 shows a task T_1 having three subtasks connected by arrows (consider these T_{11}, T_{12}, T_{13} not labeled in figure 3). After completion of the first subtask T_{11}, an event is pushed to the second subtask T_{12}, and similarly for the third subtask T_{13}. The set of three subtasks of T_1 has a single deadline and period $P_1=D_1$. In this example, this task T_1 is physically placed on three distinct processors connected via a bus or a LAN. This example explains a single task. The system is then composed of multiple tasks, each task T_i composed of one or more subtasks placed on one or more physical processors, and with communications proceeding in possibly different directions among the processors.

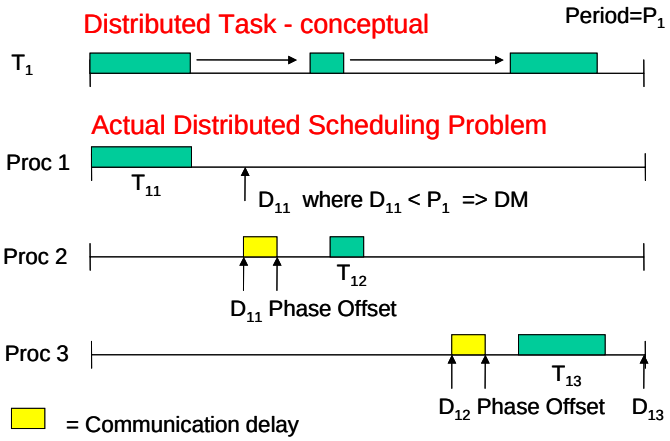


Figure 3: Schedulability Analysis for Deadline Monotonic with Phase Offset

This distributed scheduling problem can be modeled as an end-to-end scheduling problem. To provide scheduling support for the above distributed scheduling problem, VEST implements a scheduling analysis that we call *Deadline Monotonic with phase Offset* (*DM/Offset*). The assumptions of DM/Offset are (periodic, distributed).

If the design matches the assumptions, DM/Offset assigns intermediate deadlines $\{D_{ij}\}$ (e.g., D_{11}, D_{12} and D_{13} in 0) for the subtasks $\{T_{ij}\}$ of each task T_i, and accounts for the worst-case network delay t_c. The first subtask T_{i1} has a start time at the beginning of its period and a deadline less than its period; the subsequent subtask have a static phased offset equal to the deadline of its previous component plus t_c. (The static offset requires delaying the release of a subtask T_{ij} if its predecessor T_{ij-1} finishes earlier than its deadline.) The deadline of the last subtask equals the deadline of the whole task. If every subtask T_{ij} meets its intermediate deadline, the whole task meets its deadline D_i. Consequently, the distributed schedulability analysis is reduced to the analysis of each node independently with phased offset.

For schedulability analysis on each node, we employ Audsley's priority assignment and analysis algorithm [7], which provides an optimal priority assignment and feasibility test algorithm for static priority tasks with arbitrary start times (phase offsets) on a single node. It is different from Rate Monotonic and Deadline Monotonic priority assignment schemes, which assume that tasks must be released simultaneously, i.e., without considering the start times (phase offsets). The current DM/Offset analysis takes a simple approach that evenly divides the deadline of each task as the intermediate deadlines of its subtasks.

While the scheduling algorithms themselves are not novel, the extensible architecture of our scheduling tool allows us to incorporate existing schedulability analysis techniques

that handle static and/or dynamic offset in distributed real-time systems. Further, these algorithms acquire the task models and hardware models automatically from VEST itself. This facilitates repeated analysis.

3.2 Memory Footprint Check

The memory footprint check is used to verify whether there is enough physical memory to support the system software. Insufficient memory can cause serious problems when the system is deployed. There are two parts to the check. The first part of the check is concerned with main memory. Here, a sum is done of the memory needed by all the software components, and the available physical memory (RAM) provided by the hardware. The check verifies whether there is enough physical memory in the system for the software components defined. The second part of the memory footprint check involves non-volatile memory (NVM). Similar to the first check, this check verifies that there is enough NVM available in hardware as needed by the software components of the system.

3.3 Event Check

Components communicate with each other by sending events through event channels or paths. The event check iterates through all components and makes sure that every event supplier has an event consumer corresponding to it and every event consumer has an event supplier corresponding to it. Mismatches in the event channel are automatically identified. Also, circular event dependencies can be checked by going through the event channel.

3.4 Buffer size check

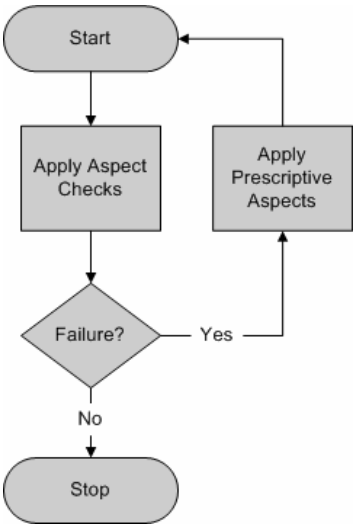
The buffer size check is used to make sure that there are no buffer overflows during communication between software components. In our design, every component has a buffer to temporarily hold event messages received from other components before they are processed. The size of a buffer needed by a component to avoid overflow is based on four parameters – the number of event suppliers, event supplier's supply rate, event consumer's consume rate and the size of the event message. The event supply and consume rate vary among different components in the system. Also, different events have different message sizes. We can calculate the size of the buffer needed by a particular component by summing of the sizes of the buffers needed for the event messages it receives from each of its event suppliers. Each event buffer size is calculated as follows

```
[(Supplier Event Supply Rate)/(Consumer Event Consume Rate)]*(Event Message Size)
```

4. Prescriptive Aspects

Prescriptive aspects (written in VPAL) are reusable programming language independent advice that may be applied to a design. For example, a designer can invoke a set of prescriptive aspects in the library to add a certain security mechanism *en masse* to an avionics product. This view of prescriptive aspects can be considered codifying expert advice. Prescriptive aspects also support system-wide modifications that are performed in an easy to change manner, and then results in a complete and consistent change to the system design.

Aspect checks and prescriptive aspects work in a complementary way. Aspect checks examine the system for hidden crosscutting dependencies while prescriptive aspects are applied to modify the system as directed by the designer, e.g., if the aspect check determines a deficiency. Their relationship is described in following diagram. Both aspect checks and prescriptive aspects are implemented as interpreters in VEST.



Suppose a given system has only periodic tasks and a change is made to add aperiodic tasks. A particular aspect check we have implemented is to identify all those components that had previously assumed that no aperiodic tasks would exist. This check detects that the scheduling algorithm also has to change (assuming that the original real-time scheduling algorithm only addressed periodic tasks). Developers are presented with the information and then must make the proper changes to the design (e.g., invoke a prescriptive aspect to add sporadic server to the scheduler) using prescriptive aspects described in the next subsection. They can then re-run this aspect check to insure that the problem no longer exists.

Prescriptive aspects have two major roles: as a system design modification tool, and as an application of expert advice obtained on previous domain specific implementations. In this section we consider each of these in turn. We then discuss the concept of hierarchies

of prescriptive aspects which are useful for both types of prescriptive aspects.

4.1 System Design Modifications

Prescriptive aspects are *advice* that may be applied to a basic functional design. This encourages a developer to design in a functional manner and then consider the non-functional aspects. This separation of concerns makes design easier. For example, a designer might create the functional modules for navigation of an aircraft and then apply advice to support real-time performance and security. Overall, prescriptive aspects support a widespread global change in the design in a complete and consistent manner by simply defining new advice or using pre-declared advice and applying it to your design. This prevents bugs where (without this support) the changes required are only made in some of the requisite places. Also implied by this advantage is that re-applying different advice can be done simply and aspect checks and schedulability analysis can be re-run automatically. This facilitates looking at multiple competing design options, thereby resulting in more effective final designs.

To change the system design, prescriptive aspects can adjust properties in the reflective information (e.g., change the priorities of a task or the replication levels of a software component). It can also add/delete components or interactions between components. When the properties of a component are changed, the associated code of this component is marked as inconsistent until it is changed to match the design.

To better understand the qualitative benefits of prescriptive aspects consider the following examples which are easy to implement with prescriptive aspects. After designing the basic system, one step towards achieving fault tolerance can be addressed by a prescriptive aspect that *makes 2 copies of all data of type waypoint_data.* A designer might also want *all data of type pilot_actions to be logged.* In addition, it is easy to specify that *all data of type Y (no matter where it is in the system) should be encrypted with a particular encryption scheme.* Many other examples can be given for non-functional categories of modifications relating to security, persistence, locking, real-time and reliability.

Normally, prescriptive aspects are used to modify the basic design. However, since the prescriptive aspect language has a create statement, prescriptive aspects can, by themselves, implement the entire basic design plus changes to it. While we have not yet investigated this feature in detail, building a system this way would be very flexible since even the basic design would be easily re-done. With this feature it is also possible to construct a subsystem or infrastructure with the prescriptive aspect language and then import that subsystem or infrastructure. For example, the design of an OS for a set top box can be designed using prescriptive aspects, then that OS infrastructure could be added to a product simply by executing the prescriptive aspect.

4.2 Expert Advice

When advice is deemed important and potentially usable on more than one project, then that advice can be generalized and placed in a global (for this application domain, e.g., avionics) prescriptive aspect library by the lead designer. VEST supports reusing such prescriptive aspects by organizing them into a prescriptive aspect library. Prescriptive aspects will not be permitted into the prescriptive aspect library unless it meets with the approval of the system administrator. The requirements include sufficiently general, parameterized, complete English description, meaningful constraints specified, and relating to non-functional properties.

One way to use the expert advice is as a collection of ideas from previous projects that might be applicable. For example, a developer can walk through all the library advice and determine if they are appropriate. After designing a functional avionics product, a developer may browse through these expert prescriptive aspects for security, real-time performance, fault tolerance, and persistence. For each category they can determine if any of the advice should be applied directly or that they need to create similar advice for their particular project. This browsing can aid in producing a more complete and tailored design and when specific advice is already in the library it is easy to apply.

Also, advice can be grouped in such a way to support implementing a wide reaching concept, such as improved computer security. Under general security advice notion, there might exist a group of prescriptive aspects that relate to denial of service, encryption, and authentication. Applying the high level advice applies the entire group.

4.3 Hierarchies of Advice

Regardless of how prescriptive aspects are added to a design there can be a need for hierarchies of advice. In some cases it may be necessary to apply to a design a set of seemingly “unrelated” aspects in some order. To support this feature, the developer has the capability to describe precedence constraints among the aspects. More importantly, the same mechanisms can be applied to create a “related” set of changes to effect a global change to the system (as described above for the security example). In order to make high level changes to a design (e.g., in regard to security, fault tolerance, reliability, and performance), it is usually necessary to make a set of “related” and more specific changes. For example, there can be a group of advice in the prescriptive library that supports a secure avionics system. This advice may encompass a collection of changes that includes encrypting certain types of communication, adding intrusion detection changes, adding modifications that prevent or minimize denial of service. The mechanisms in VEST support this type of design where the root of the hierarchy can imply changes needed for security, and the rest of the tree contains the specific modifications required.

5. VEST Prescriptive Aspect Language (VPAL)

5.1 Design Philosophy

VPAL enables users of VEST to specify their prescriptive aspects. The syntax of VPAL is specific to the VEST entities that specify components, their attributes, and interactions between components. Ease-of-use and modification power are the driving forces behind VPAL's design. VPAL allows the specification of modifications using a simple yet powerful syntax. Because of this, VPAL is a language with no data type declarations, procedures, control flow, loops and classes. VPAL's syntax consists of just four key statements. It would take a few minutes for a novice programmer to understand VPAL and be able to write prescriptive aspects. The power of VPAL's syntax can only be fully realized through its use. The evaluation section presents concrete examples of the time saved in by designers using prescriptive aspects written in VPAL.

VPAL is similar to SQL except that the data set being operated on is sets of components rather than sets of rows from a table. It is not a procedural, functional, object-oriented or even aspect-oriented programming language. It is intended to be specifically used in the VEST tool for easily creating prescriptive aspects.

5.2 Separation of Concerns

As mentioned earlier, prescriptive aspects change a design by adjusting properties in the reflective information of components and/or by adding/removing components from the design. VPAL explicitly separates the concerns of collection, operation, addition and removal of components. Four key statements in the language, *Get*, *Set*, *Create* and *Delete* enable this separation of concerns. Each of these concerns plays an important role in fulfilling the objective of prescriptive aspects and they are described in detail below. The full BNF specification of the VPAL syntax can be found in Appendix A.

5.2.1 Collection

A Collection is defined as a set of components from a system design. A collection enables a designer to represent a cross section of the design based on the properties of components or the relationships between them. This is essentially the value of collection as it enables a designer to quickly and easily identify components to be modified which would have otherwise taken much manual search time. The *Get* statement in VPAL implements this feature. It assigns the collection to a variable for later use. For example, the GET statement

```
GET SWComps = (CT == SoftwareComponent);
```

finds all components whose component type (CT) is "SoftwareComponent" in the design and assigns this set to a variable called "SWComps". The right side of the statement

specifies the search criteria. In this case, we used the component property of type as our search criteria, but in general, it can be any component property such as type, name or any of the extensive list of attribute values found in the reflective information of a component. Search criteria can also be combined into compound statements with boolean operations *AND*, *OR* and *NOT*.

5.2.2 Operation

An Operation involves changing a design on previously gathered collections. An operation enables the weaving of user-defined changes into a design. Operations on collections are performed with the *Set* statement that adjusts the properties in the reflective information of the collection. For example, the SET statement

```
SET SWComps.(PN = MemoryNeeded, PV = 0);
```

initializes the property (attribute) name (PN) “MemoryNeeded” of all components in the “SWComps” collection to a property value (PV) of zero.

5.2.3 Addition and Deletion

Addition and removal of components are self-explanatory. These commands enable users to weave changes into a design. Addition of components could also potentially be used to create large designs from scratch. The *Create* statement in VPAL adds a set of components to the design and assigns this set to a variable for later use. For example, the CREATE statement

```
CREATE DispComp.(CT = SoftwareComponent,
     CN = MyDisplayComponent,
     PN = ComponentType,
     PV = BM__DISPLAY_COMPONENT);
```

creates a display software component with a component name (CN) of “MyDisplayComponent” and assigns it to variable “DispComp”.

The *Delete* statement removes previously defined collections from the design. For example, the DELETE statement

```
DELETE DispComp;
```

deletes from the design the components defined in the “DispComp” collection.

5.3 Multi-line Semantics

VPAL supports multi-line semantics. This means that each prescriptive aspect can contain multiple lines of instructions. Each instruction is one of the four statements that were described above. The multi-line semantics of VPAL allows a user to define and operate on multiple collections within the same prescriptive aspect.

For example, suppose we wanted to apply the following prescriptive aspect to a distributed avionics system being designed in VEST:

```
Double the memory needed for all device software components
  - and -
change all display software components to use double buffering
```

Using the multi-line semantics of VPAL, we could specify this prescriptive aspect as

```
[1]  GET SwComp = (CT == SoftwareComponent);

[2]  GET DevComp = SWComp.(
       PN == componentType,
       PV == BM__DEVICE_COMPONENT);

[3]  GET DispComp = SWComp.(
       PN == componentType,
       PV == BM__DISPLAY_COMPONENT);

[4]  SET DevComp.(PN == MemoryNeeded, PV = PV * 2);

[5]  SET DispComp.(PN == DoubleBuffered, PV = 1);
```

This prescriptive aspect contains two different cross-sections of the design of interest to the designer. One contains all device components (line 2) and the other contains all display components (line 3). The designer then modifies each set according to the change desired (lines 4 and 5).

While VPAL is simple, the downside of simplicity is that the expressive power of the language is limited sometimes resulting in redundant code. For example, consider a design with a large number of software components that are sub-classified into many software component types. Suppose we wanted to write a prescriptive aspect to initialize several of the attributes of these software components to different values by type. The code would contain redundancy for a design with a large number of software component types. This redundancy could be eliminated with loops in VPAL. VPAL can be extended to allow loops and other programming language concepts such as control flow, procedures, inheritance, overriding, and so on but we have not found it necessary for embedded systems of small or moderate size.

6. Case Studies and Experiments

6.1 Case Study I: Composition and Analysis using Product Scenario 3.1 (Basic MP)

The purpose of this case study is to demonstrate the effectiveness of the ideas incorporated in VEST. To do this we applied VEST to the design and composition of a portion of a distributed avionics system that is based on the Bold Stroke middleware. In this avionics system, a pilot control component measures coordinate data periodically, then sends its coordinate data to a waypoint control component. Upon receiving coordinate data, the waypoint control component calculates a new route for the plan, updates its database, and sends that new route to a display component. This avionic control system is a typical example of a distributed real-time embedded system with many crosscutting concerns. In fact, this example scenario is posted by Boeing as a good scenario for evaluating design and analysis tools as Product Scenario 3.1, BasicMP (Basic Multi-Processor) Figure 4 below shows the UML diagram of the avionic system's software architecture.

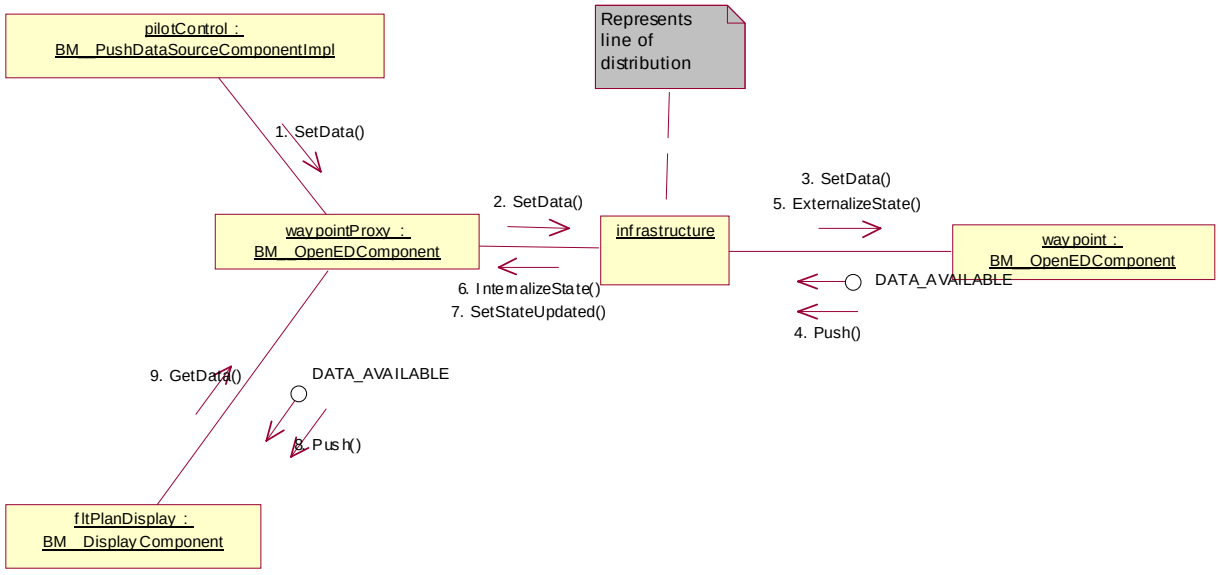


Figure 4: UML Diagram of a Pilot Control Subsystem

To better understand the case study, additional details about the application are provided: The system is composed of four first level components: pilotControl, waypointProxy, waypoint, and fltPlanDisplay. They run on the Bold Stroke middleware. The pilotControl component is an event supplier. It supplies coordinate data to the waypointProxy component at a specified frequency. WaypointProxy is a proxy representing the waypoint component and it runs on another processor. Communication is supported by the middleware service known as an event channel. Via the event channel, data originating in the pilotControl component is forwarded to the waypoint component. Likewise, the waypoint component sends the newly calculated route back to

waypointProxy. Finally, the fltPlanDisplay component gets the new route information and displays it.

6.1.1 Design the Pilot Control Subsystem

In this case study, the developer first creates the system using abstract components. After the abstract specification has been performed, the system design might look as shown in Figure 5.

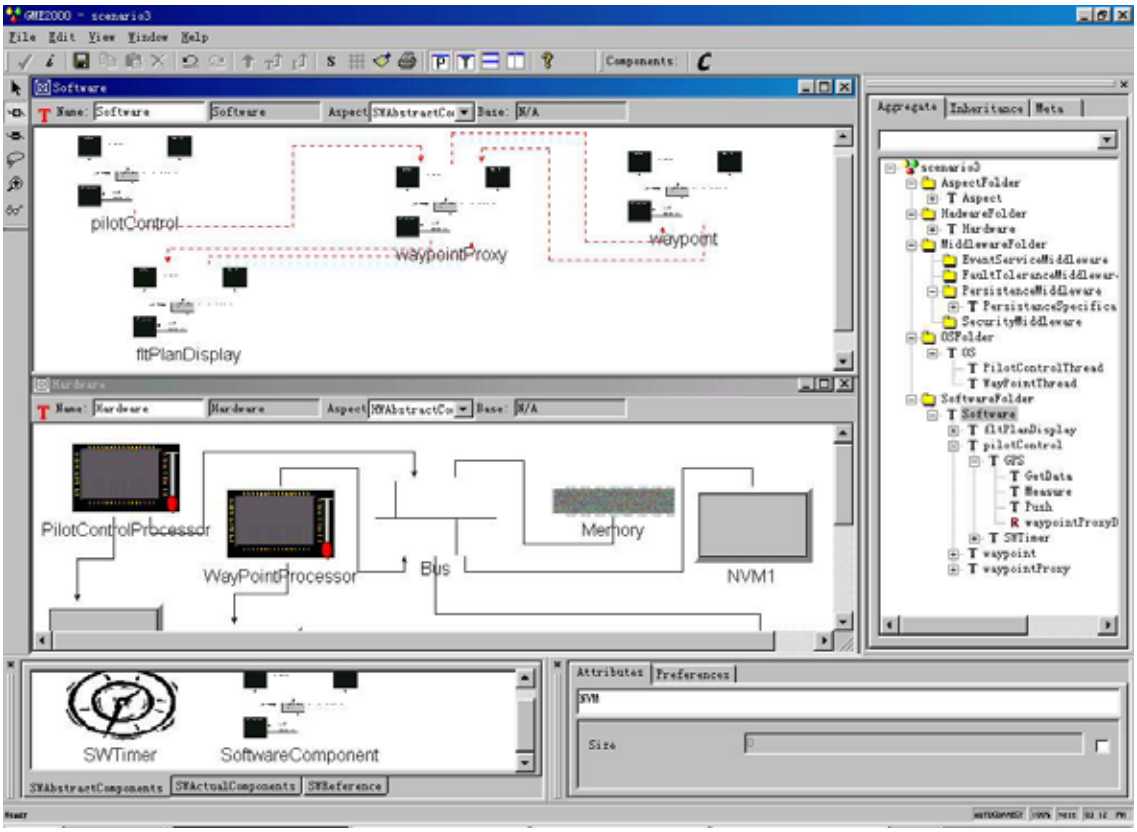


Figure 5: VEST model of a pilot control system

In the above diagram there are two layers shown. One layer is the software layer. This layer (see the top panel of figure 5) has basic four components: pilot control, waypointProxy, waypoint, and fltPlanDisplay. The high-level interaction of these components is shown by the dashed lines. By high-level interaction we mean that if there is any event propagation from one component to another, then these components are connected by an arc. Direction of a connection shows the flow of events. A second panel in the picture shows the hardware layer. In this example, the system is deployed in a distributed environment. It contains two processors: a pilot processor and a waypoint processor. They are connected via a bus interconnect. Also, the system has two non-volatile memory units and one volatile memory unit. What are not shown in the diagram are the OS, Aspect, and middleware layers. The components of these layers can

be viewed from the browser menu shown on the right-hand side of figure 5. In the OS layer, we have two threads: a Waypoint thread and a PilotControl thread. The waypoint thread is mapped to the waypoint processor and the pilot control thread is mapped to the pilot control processor. The components that run on the waypoint thread are Pilot Control, WaypointProxy and fltPlanDisplay.

The persistence service of Bold Stroke is one focus of this case study. Every application component that needs to maintain persistent data needs to create a persistence adapter that set the follow attributes of the persistence service: save_rate, is_double buffered, and track_dirtiness. The save rate specifies the frequency of the persistence thread. Is_double_buffered identifies whether the state should be saved twice or not. Track_dirtiness is a boolean variable; if true, this parameter causes the state to be persistent if the persistent object is dirty (i.e. has been modified and thus the new state needs to be saved).

Double clicking on the software components shows the methods and member variables modeled in this component. An event graph is specified at this view (VEST models systems at the method level). The specification of the method-calling graph helps in completely characterizing the systems execution and thereby provides needed data for VEST to perform interface checking and schedulability analysis. After performing these operations, the developer chooses actual components from the libraries and maps them to these abstract components. After modeling, the VEST developer makes various checks to boost his confidence in the correctness of the system.

6.1.2 Memory Footprint Check

In this case study, the first checks performed are intra-component checks. For instance, enough memory is vital for the system's performance. A memory footprint check is available in VEST. The first part of the memory footprint check is concerned with main memory. It sums the memory needed by all the components in the system, and all the available physical memory (RAM) provided by the hardware, and then checks to see if there is enough physical memory in the system. In the case study, the developer initially specified the system as follows:

	PilotControl	WaypointProxy	Waypoint	fltPlanDisplay
max memory footprint	50M	100M	300M	100M

However, the hardware memory is only of size 500M. Considering the system overhead, the memory check informs the developer of insufficient memory. The developer either adds more memory, or reduces memory consumption by modifying application components.

The second part of the memory check deals with NVRAM (e.g., EEPROM). Bold Stroke allows application programs to specify a set of data in some components to be persistent, so that important data in the system survives power failures. For the system to function correctly, sufficient NVRAM for persistent components should be provided. Our check

assures the developer that there is enough non-volatile memory to meet the system's requirement, or gives warning when not enough NVRAM is provided. In this case study, the system has two NVRAMs with a total capacity of 300 MB. The sum of the persistent objects' size is 200 MB. The persistent object is originally configured as double-buffered, which doubles the needed capacity of NVRAM to 400 MB. When invoked, the memory footprint check warns that there is insufficient NVRAM. In this case study, the designer now reconfigures the persistence adapter to single-buffered mode, and the memory check returns successful confirmation. While these checks are trivial, they are useful and demonstrate a simple cross cutting constraint. Further, these checks are enhanced in the prescriptive aspect example in the next section.

6.1.3 End-to-End Schedulability Aspect Check

The developer may then proceed to make additional checks that are more sophisticated. VEST provides an automatic schedulability analysis. After the designer completes the design of the model, he runs the schedulability analysis to check the model. This analysis requires the DM/Offset analysis because the software components are mapped to multiple interconnected processors in the model. However, the output of the schedulability analysis shows that the model is not schedulable, as depicted in the following. The beginning of the output is a method list including the period and worst-case execution time (WCET) of the methods in the CORBA components. Based on the event graph, multiple interacting methods on a same processor are grouped into a subtask, which is mapped to a thread. The second part of the output is the subtask list on each processor and its schedulability analysis results. The subtask list includes the period, WCET, and the intermediate deadline and offset of each subtask. For the initial design with a period of 400 ms, the analysis shows that processor 2 is schedulable, but processor 1 is not. Therefore, the design should be changed.

```
List of methods
MethodName MeasureLocation   Processor Processor1 Period 400 WCET 67
MethodName Push          Processor Processor1   Period 400  WCET 2
MethodName Push          Processor Processor2   Period 400  WCET 4
... ...
Subtasks on Processor2
Subtask Push Processor 2 Period 400 WCET 4 Deadline 160 Startime 81
Subtask CalculateRoute  Processor 2 Period 400 WCET 2 Deadline 320
                              Startime 241
Priority level 2 has been assigned to Push.
Priority level 1 has been assigned to CalculateRoute.
Schedulability test on Processor2 passed.
Subtasks on Processor1
```

```
Subtask MesureLocation+Push+GetData Processor 1  Period 400 WCET 102
                                    Deadline 80 Startime 0
Subtask DataReadyPush+Push+GetData+Display Processor 1 Period 400  WCET
                                                    11
                                                Deadline 240 Startime 161
Subtask GetData Processor 1 Period 400 WCET 2 Deadline 400  Startime 321
      Couldn't assign, try next
      Priority level 3 has been assigned to
DataReadyPush+Push+GetData+Display.
      Couldn't assign, try next
      Priority level 2 has been assigned to GetData.
      Couldn't assign, try next
Schedulability test on Processor1 failed.
```

Output of the schedulability check on the original pilot control subsystem with a period of 400 ms

6.2 Case Study II: Measurement of Composition Time using Product Scenario 3.2 (Multi-Rate MP)

We performed a second case study to measure the benefits of VEST in composing distributed avionics systems. The performance metric is the time it takes to compose (including design, implementation via composition, and testing or analysis) an avionics product scenario to achieve end-to-end distributed real-time schedulability. This experiment was accomplished in a very limited situation. One expert from Boeing performed the experiment using their current approach. And one grad student performed the experiment using VEST. For each person we timed the various steps involved with this experiment. Since this is a single experiment with many potential issues, the results are not definitive. However, we believe that the results are representative and discuss how they might generalize to a larger experiment.

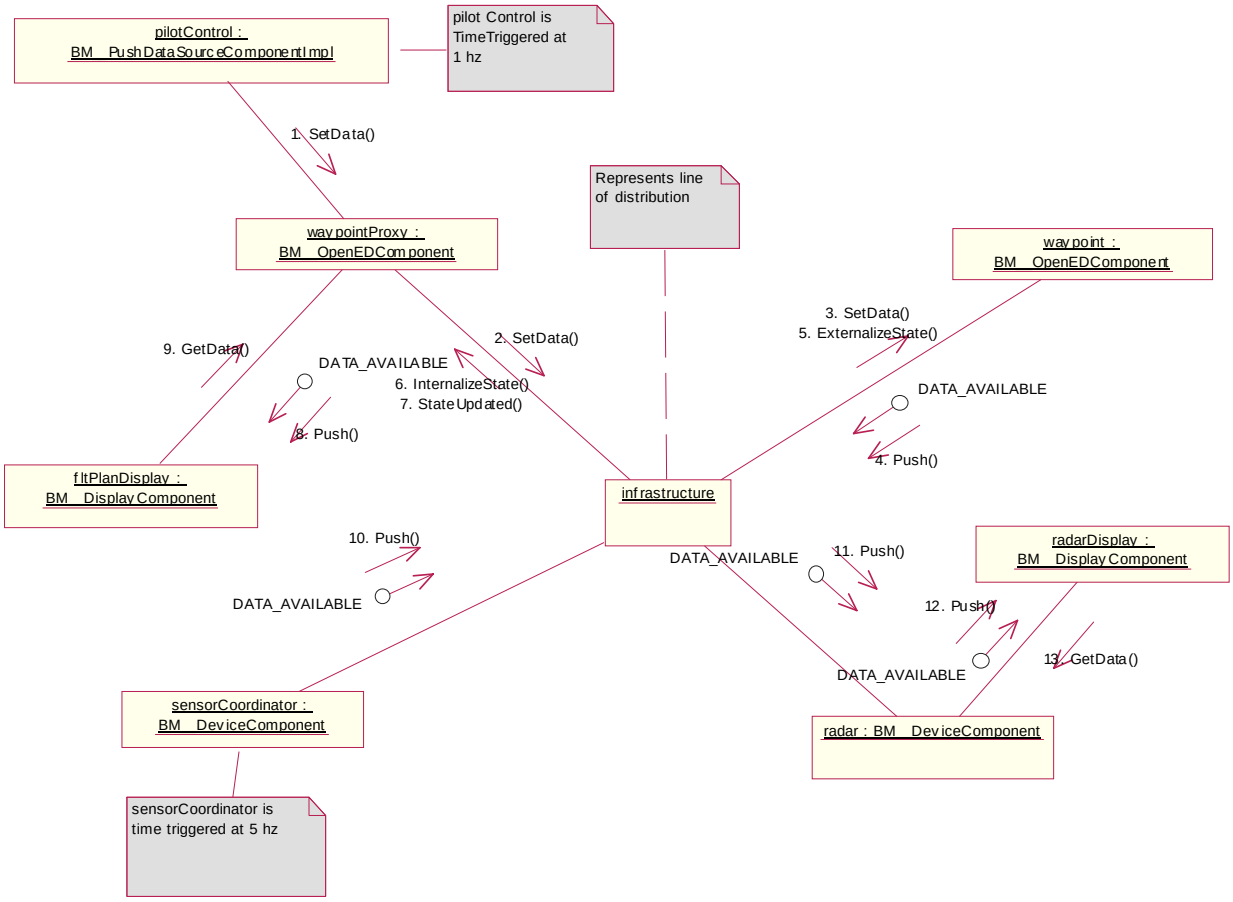


Figure 6: Multi-Rate Multiprocessor Scenario

The experiments used Product Scenario 3.2 (Multi-Rate Multi-Processor) as a target system to be composed. The scenario represents that portion of an avionics system that displays waypoint and radar data and is published by Boeing as a typical subsystem to facilitate research that is applicable to real world problems. The waypoint data can be changed by the pilot and the radar data is produced at a 5hz rate by the radar device. The sensor coordinator notifies each logical sensor of when its data should be updated.

This scenario is initially triggered by an interval timeout that is consumed by the pilotControl component. Upon receipt of this event, the pilotControl pushes data to the waypointProxy via the Set operations in the proxy's facet. The waypointProxy then forwards this call via the Infrastructure component to the waypoint component. The waypoint then updates its state and issues a *Data Available* event. That event causes the Replication Service to extract the state from the waypoint and send it to the waypointProxy. The waypointProxy internalizes this state and issues its own *Data Available* event. The proxy's event is consumed by the fltPlanDisplay component that gets the data from the proxy and displays it.

The baseline toolset for comparison includes Rational Rose and Quantify both of which are currently used in Boeing's product development. The UML models of all Bold Stroke components were available in Rational Rose before the experiment started. WCET of all used components were also available in the library before the experiment started. An expert at Boeing used the following process to compose Product Scenario PS 3.2:

1. Design PS 3.2 by integrating the UML models of existing components in Rational Rose.
2. Implementation: Program the design by connecting existing Bold Stroke components in C++ through the Bold Stroke event service.
3. Testing: Run the implemented system to check for timing violations. If any timing violations are detected, go back to step 1; Otherwise, the composition is completed.

At UVA, a graduate student familiar with VEST used VEST to compose the same product scenario. The VEST experiment included the following steps:

1. Design PS 3.2 in VEST using component libraries.
2. Scheduling analysis: Run the VEST scheduling tool to assess the schedulability of the design (without implementing the system). If the analysis shows that the design is not schedulable, go back to step 1. Otherwise, go to step 3.
3. Implementation: Program the VEST design.

Both VEST and the baseline experiments included two iterations of composition. Initially, the system was designed on a single-processor platform. Since the single-processor design turned out to be unschedulable, a new composition was needed. A new processor was added to the system and a distributed version of PS 3.2 was composed by moving several components to the new processor. The distributed version was found to be schedulable. The VEST scheduling tool can automatically identify the applicable scheduling analysis that matches the system characteristics. Maximum Urgency First (MUF) scheduling analysis was automatically invoked for the single-processor design, and the DM/Offset scheduling analysis was automatically invoked for the distributed design.

We measured the total composition time as well as the time that each step took in both experiments. The results are summarized in **Table 1**. We used *X.i.k* to represent the k^{th} step in the i^{th} iteration of the *X* experiment, where *X*=*V* refers to the VEST experiment and *X*=*B* refers to the baseline experiment.

VEST			**Baseline**		
Step		Time (min)	Step		Time (min)
V.1.1	Design: single processor	40	B.1.1	Design: single processor	25
			B.1.2	Implement: single processor	75
V.1.2	Scheduling analysis: single processor	1	B.1.3	Test: single processor	30
V.2.1	Re-design: distributed	25	B.2.1	Re-design: distributed	90
			B.2.2	Implementation: distributed	105
V.2.2	Scheduling analysis: distributed	1	B.2.3	Test: distributed	20
	Implementation: distributed	105			
Total Composition Time		172	Total Composition Time		345

Table 1: Measured Time with VEST and Baseline in Case Study II

Our measurement showed that VEST effectively reduced the total composition time of PS 3.2 by 50%. Analyses on the time spent on each step shows two key advantages of VEST compared to the baseline:

- Reduce the rounds of implementations: Scheduling analysis enables VEST to drop wrong (unschedulable) designs without implementing the system. In this case study, the scheduling analysis showed that the single-processor design was unschedulable. Hence the VEST user avoided implementing the single-processor composition (Step B.1.2) and saved 75 min. Compared to the baseline, this reduced the total composition time by 22%. Note also that in VEST the scheduling is a rigorous analysis and in the standard approach it is only done via testing which is more error prone.
- Replace time-consuming testing with quicker analyses: Two schedulability analyses (Steps V.1.2 and V.2.2) in VEST took a total of only 2 minutes, compared to a total testing time of 50 minutes (Steps B.1.3 and B.2.3) in the baseline experiment. This saved the VEST user 48 min and reduced the composition time by 14% compared to the baseline.

While this case study focused on the scheduling part of VEST, we should note that both of the above benefits are also applicable to other aspect checks of VEST. Aspect checks enable developers to detect crosscutting composition errors at design time and let

developers correct system designs without implementing them. Since crosscutting dependencies are often non-obvious and difficult to find through test, explicit design-time checks can save significant amount of testing time.

While the initial experimental results on this relatively simple scenario are very encouraging, we expect VEST to save even more time in more complex systems with larger number of components. To show this, we now give a simple analysis. Let us assume that

- composing a system requires N iterations;
- implementing a design from VEST and the baseline both take T_i min;
- the average time for each testing is T_t min.
- The average time for each design in VEST and the baseline are both T_d;
- The average time for each analysis in VEST is T_a. Since analyses take much less time than testing, $T_a << T_t$.

The total composition time with the baseline is $T_{base} = N(T_d+T_i+T_t)$, and the total composition time with VEST is $T_{vest} = N(T_d+T_a)+T_i$. It follows that the time saved by VEST is $\Delta T = (N\text{-}1)T_i+N(T_t\text{-}T_a)$. While this analysis is somewhat simplified, the general conclusion is that the more design iterations (N) a system need, the more time an aspect tool like VEST can save in the composition process. Since complex systems usually involve more crosscutting dependencies and require more iterations, we expect VEST to scale much better than the baseline in such systems. We plan to test VEST in a bigger product scenario (with more than 400 components) to verify the scalability of VEST in large systems.

6.3 Case Study III: Focusing on Prescriptive Aspects using Product Scenario 3.3 (Concurrency MP)

In this section, we demonstrate the benefits of prescriptive aspects through a case study. We apply prescriptive aspects to the design of an avionics system, which is based on the Boeing Bold Stroke platform. We show how prescriptive aspects support system modification, provide expert advice, and save 69% of design time. This savings is obviously quite conspicuous. But it should be noted that the specific scenario was set up to result in scheduling and memory system errors that would require a great deal of post-design coding and testing to discover without the use of the VEST design time analysis capabilities. In the situation that a system did not contain such errors, the time savings from using VEST would be greatly reduced. The more VEST-detectable errors a system has – the greater the benefit of using VEST.

The baseline toolset for comparison includes Rational Rose and Quantify which are both currently used in Boeing's product development. The UML models of all Bold Stroke

components were available in Rational Rose before experimentation started. The worst-case execution times (WCET) of all components used were also available before experimentation began.

Figure 7 below shows the UML diagram of the software architecture of P.S 3.3 (Concurrency Multi-Processor). This system corresponds to a navigation type function on an aircraft. The aircraft maintains a list of waypoints (points to fly the aircraft to).

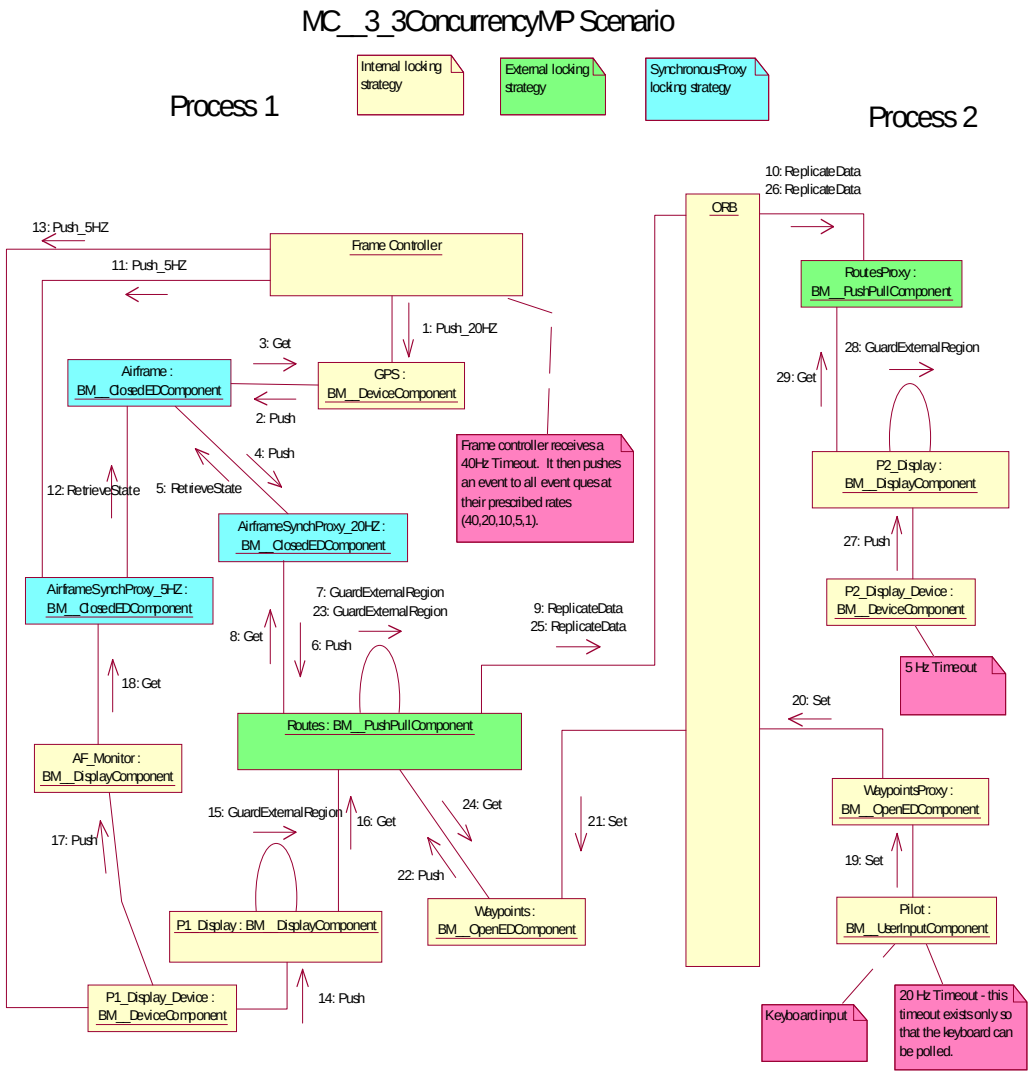


Figure 7: Concurrency MP Scenario

Waypoints are selected in groups to form routes (a series of points to fly the aircraft to, one after the other). The pilot can modify the waypoints to change the current route of the aircraft. In addition, GPS sends location information to the system periodically. The current waypoint and current aircraft position are displayed periodically.

This navigation system is a typical example of a distributed real-time embedded system with many crosscutting concerns. Such concerns include real-time schedulability as well as event channel, memory and buffer requirements. These and many other concerns are critical to the overall system. We use aspect checks to identify them and prescriptive aspects to modify them if they do not meet the system requirements.

Throughout our experiment, prescriptive aspects are used for two primary purposes: system modification and expert advice.

6.3.1 System Modification

First in this experiment, the designer started to design an avionics system in VEST based on product scenario 3.3 provided by Boeing. He composed the system using the components from the VEST component library. Afterward, he assigned different values to attributes such as memory size, buffer size, WCET, and period to the components accordingly.

After running the memory footprint aspect check however, the designer found out that the amount of memory allocated in hardware was smaller than required by the software components. Instead of modifying the attribute values (named MemoryNeeded) of the components manually, the designer decided to use a prescriptive aspect. He executed the following prescriptive aspect, which reduces by half the memory allocated to software components of type BM__DISPLAY_COMPONENT.

```
GET A = (CT == SoftwareComponent) AND
     (PN == componentType, PV ==
BM__DISPLAY_COMPONENT);
```

Then he re-ran the memory footprint check and it passed. This saves time over modifying the system parameters manually and is more accurate.

After checking the memory allocation, the designer checked the schedulability of the system design by running the schedulability aspect check. The check failed because in this case, event suppliers in the system were specified to have too high a WCET value that caused tasks in the system to miss their deadlines. In general, there can be several factors that cause a schedulability test to fail such as insufficient task period or high WCET value. Again, instead of modifying all these parameters manually, the designer modified the system automatically by executing the following prescriptive aspect.

```
GET SW = (CT == SoftwareComponent);
GET ES = (CT == EventSupplied);
GET EC = (CT == EventConsumable);
GET ContES = SW[$ONEONE,$DR=$CONT]$ES;
GET ContEC = SW[$ONEONE,$DR=$CONT]$EC;
GET MappedES = $ES[$MANYONE,$DR=$CONN]
EC;
SET MappedES.(PN=WCET, PV=10);
```

This prescriptive aspect collected the event supplier components that are *contained* in software components and are *connected* to event consumer components, and set their WCETs to 10ms. After making this modification, the schedulability test passed. Of course, these modified components must be reprogrammed to meet this new WCET.

6.3.1.1 Expert Advice

Prescriptive aspect can be used to provide expert advice on the design of a system. Expert advice in this context is generic advice that applies to various scenarios sharing the same meta-model. Usually expert advice is stored in a library. Designers can retrieve the expert advice from the library and reuse the advice by applying them to every relevant scenario conforming to the same meta-model.

In this case study, we used assignment locking strategies to components as an example of expert advice. There are three kinds of locking strategies used by components in the Bold Stroke platform: internal, external and synchronous proxy. The internal locking strategy requires a component to lock itself when data is modified. An external locking strategy requires the user to explicitly acquire a component's lock before accessing its data and release the lock when finished. The synchronous proxy strategy requires the use of cached states. Knowledge of such locking strategies is generic and applies to all Boeing OEP product scenarios. Therefore, we put this particular prescriptive aspect into the general expert advice library. When a designer wants to apply this set of locking strategies to his design, he can choose the prescriptive aspect from the library and execute it.

The internal locking strategy:

```
GET SW = (CT == SoftwareComponent);
SET SW.(PN=lockingMode, PV=INTERNAL);
```

The external locking strategy:

```
GET PushPull =(CT==SoftwareComponent)
    AND (PN==componentType,
    PV==BM__PUSH_PULL_COMPONENT);
GET EC = (CT == EventConsumable);
GET PushPullMappedEC = $PushPull
    [$ONEMANY,$DR=$CONT] EC;
SET PushPullMappedEC.
```

And synchronous-proxy locking strategy:

```
GET SW = (CT==SoftwareComponent);
GET EC = (CT==EventConsumable);
GET Timers = (CT==SWTimer);
GET SWMappedEC = $SW
    [$ONEMANY,$DR=$CONT] EC;
GET SWMappedTimer = $SW
    [$ONEONE,$DR=$CONT] Timers;
GET SynchProxy = $SWMappedEC
    [$ONEMANY,$DR=$CONN]
    SWMappedTimer;
SET SynchProxy.(PN=lockingMode,
```

By default, we assume every software component uses internal locking. A “PushPull” software component is defined as one that updates its values (by pulling or getting data from its suppliers) when it receives an indication (through a push or set). According to our application rules, any PushPull software component that has one or more data suppliers must use external locking. This is what is coded in the external locking prescriptive aspect. Finally, any component that receives data from more than event channel, each running on different timers in the system should use synchronous-proxy locking as indicated by the last prescriptive aspect.

By applying this expert advice, we assign different locking strategies to all the software components in the system. This prescriptive aspect is stored as expert advice in the library. Using prescriptive aspects for expert advice saves the designer a lot of time by automating decision-making. This is especially useful when used in designs with a large number of components and where there is a lot of interaction among the components.

6.3.2 Hierarchical Prescriptive Aspects

A simple prescriptive aspect is a self-contained entity of one or more VPAL statements. The previous sections illustrated some simple prescriptive aspects. In addition, VEST provides support for hierarchical prescriptive aspects.

Hierarchical prescriptive aspects are comprised of one or more simple prescriptive aspects with precedence constraint rules. This enables a designer to define several independent simple prescriptive aspects that can later be combined into a single compound prescriptive aspect. In addition, the designer can ensure that when the compound prescriptive aspect is executed, there is a guarantee over the order of execution of the constituent simple prescriptive aspects.

We used a hierarchical prescriptive aspect to perform system initiation in our experiment. We defined independent prescriptive aspects to initialize the memory requirements of the system, the buffer size allocation, real-time properties of components such as WCET and the locking strategies to be used by different components of the system. In the interest of space, we do not show these prescriptive aspects here. By combining these prescriptive aspects into a single hierarchical prescriptive aspect, we were able to precisely define how our design should be initialized before being deployed.

6.3.2.1 Experimental results

We performed an evaluation to measure the benefits of prescriptive aspects in composing distributed avionics systems. The performance metric is the time it takes to compose (including design, implementation via composition, and testing or analysis) an avionics product scenario to achieve end-to-end distributed real-time schedulability, memory allocation, buffer size assignment and locking strategy assignment. This experiment was accomplished in a very limited situation. An expert from Boeing performed the experiment using their current approach, and a researcher from UVa performed the experiment using prescriptive aspects in VEST. For each person we timed the various steps involved with the experiment. Since this is a single experiment with many potential issues, the results are not definitive. However, we believe that the results are representative and are consistent with other tests performed earlier on other product scenarios.

The baseline comes from the time estimates for Boeing to build, analyze and validate an avionics system conforming to product scenario 3.3, while VEST uses prescriptive aspects to do the same work.

VEST			Baseline		
Step		Time (min)	Step		Time (min)
V.1.1	Design:	128	B.1.1	Design	280
	Memory check	1	B.1.2	Memory check	20
	Fixing memory problem using VPAL	20		Fixing memory problem	80
V.1.2	Scheduling check:	1	B.1.3	Timing Test	30
	Fixing scheduling using VPAL	15		Fixing scheduling	110
	Scheduling check	1		Timing Test	20
	Scheduling analysis: distributed	1		Test: distributed	20
	Assign locking strategies using VPAL	1		Assign locking strategies	30
	Implementation:	320		Implementation	960
Total Composition Time		488	Total Composition Time		1550

Table 2: Comparison between UVa and Boeing data

From the Table above all steps in the design process are faster with VEST. Overall, the VEST approach saved 69% of the time needed to design and implement a distributed avionics system. Since the memory and real-time scheduling analysis are automatic, the VEST tool should save even more time both (i) when used for larger systems, and (ii) when designers wish to attempt multiple competing designs. For example, suppose a particular design solution, shown to meet the requirements, had 3 processors, 1 MB of memory and various amounts of replication for different data types. The designer might consider removing a processor and modifying some of the replication and re-run the analysis. Re-running the analysis is very fast and each tradeoff-analysis cycle improves the time gains of using VEST. If the new system still meets the requirements, then the designer has competing solutions to choose among.

7. Deliverables

There have been 4 major releases and 2 minor releases of VEST. These releases are numbered starting at 1.0 and ending at 5.0. The delivery timeline for these releases is shown below.

	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
2002				1.0								
2003				2.0			3.0				3.1	
2004	4.0					4.1			5.0			

Each of these releases is outlined below along with the features that were included.

7.1 VEST 1.0

- VEST meta-model implemented in GME
- Features
  - RM and EDF RT-Scheduling Check
  - First Version of Prescriptive Aspect Interpreter
  - Automated Inputs (Linkage with MOBIES)
    - ACL map to VEST (components)
    - IIF map to VEST (execution time and dependencies)

7.2 VEST 2.0

- RT-Scheduling
  - Deadline monotonic with Phased offset
- VEST to XML Configuration Mapper
  - Modified VEST model to include home, receptacles, distributed roles, persistence and concurrency
  - Produce configuration file

7.3 VEST 3.0

- Executable and source code made available with InstallShield
- New/Enhanced Features
  - Buffer-size Aspect Check
  - Memory-size Aspect Check
  - RT-Scheduling Check
    - Enhanced to support centralized and distributed robust scheduling
- New hardware components added to library
- 50 page User's Manual supplied with release

7.4 VEST 3.1

- New/Enhanced Features
  - Event Aspect Check
  - RT-Scheduling Check

        - New Scheduling API provided to support other Technology Developers

7.5 VEST 4.0

- New/Enhanced Features
    - Interpreter for new VEST Prescriptive Aspect Language (VPAL) implemented
- Updated User's Manual

7.6 VEST 4.1

- VEST Metamodel Improvements
    - Resolved data type, naming and redundancy inconsistencies
- New/Enhanced Features
    - VPAL
        - Create Statement implemented
        - More robust and user-friendly interface implemented
- New event channel model implemented
- New GUI icons provided

7.7 VEST 5.0

- Initial Quality of Service Capabilities

8. Summary

When building embedded systems from components, those components must interoperate, satisfy various dependencies, and meet non-functional requirements. The VEST toolkit can substantially improve the development, implementation and evaluation of these systems. The toolkit focuses on using language independent notions of aspects to deal with non-functional properties, and is geared to distributed embedded system issues that include application domain specific code, middleware, the OS, prescriptive aspects, and the hardware platform. The VEST tool has been implemented and used on three case studies, two of which are described in this paper. The case studies (i) qualitatively demonstrate the benefits of our tool and (ii) include quantitative data that show a savings of over 50% in design and analysis time. Overall, a main advantage of our tool is that it has the potential to address the most difficult parts of component composition, the hidden crosscutting dependencies including overall, distributed real-time analysis.

Boeing is investigating the use of this tool as an integral part of a larger reuse library for avionic components. A very important advantage of VEST is using it as a repository for capturing reflective design information of system components. However, building and maintaining such a repository requires a significant commitment. Additional follow-up testing is needed to verify that VEST is scalable for use in typically large avionics systems. Hopefully, additional resources can be obtained to incorporate Quality of Service design capabilities into VEST, which will make it an even more valuable design tool.

9. References

[1] Stankovic J., Nagaraddi P., Yu Z., He Z., Ellis B., “Exploiting Prescriptive Aspects: A Design Time Capability”, *4th ACM International Conference on Embedded Software (EMSOFT)*, Pisa, Italy, September 2004.

[2] Stankovic J., Nagaraddi P., Yu Z., He Z., “VEST User’s Manual”, *University of Virginia Technical Report TR-CS-2004-10*, June 2004.

[3] Stankovic, J., Zhu, R., Poornalingham, R., Lu, C., Yu, Z., Humphrey, M., and Ellis, B., “VEST: An Aspect-Based Composition Tool for Real-Time Systems”, IEEE *Real-Time and Embedded Applications Symposium (RTAS)*, Washington, D.C. May 2003.

[4] Stankovic J. et al, “VEST: An Aspect-Based Real-Time Composition Tool” *University of Virginia Technical Report TR-CS-2003-07*, March 2003.

[5] Stankovic J., “VEST: A Toolset For Constructing and Analyzing Component Based Operating Systems for Embedded and Real-Time Systems”, *University of Virginia Technical Report TR-CS-2000-19*, July 2000.

[6] Virginia Embedded Systems Toolkit (VEST)
URL: http://www.cs.virginia.edu/~pnn7f/vest/

[7] N. C. Audsley, “Optimal Priority Assignment and Feasibility of Static Priority Tasks With Arbitrary Start Times”, YCS 164, Dept. Computer Science, University of York (December 1991).

Appendix A. BNF Grammar of VPAL

```
<statement_list>    =   <statement_list> <statement> ”;”

<statement>         =   <get_statement>
                    |   <set_statement>
                    |   <create_statemnt>
                    |   <delete_statement>

<get_statement>     =   GET <Variable> “=” <get_expression>
                    |       GET <Variable> “=” <Variable>“.””(“ <get_expression>“)”


<set_statement>     =   SET <Variable>“.””(“ <set_expression> “)”

<create_statement>  =   CREATE  <create_expression>

<delete_statement>  =   PDELETE <delete_expression>

<create_expression> =   <Variable> “=” “(“ STRING “,” <create_sub_expression> “)”
                    |   <Variable> “.” “(“ <Relation> “,”
                        <create_sub_ref_expression>“)”

<create_sub_expression> =  <ObjectType> “=” <Value>  “,” <create_sub_expression>

            |   <ObjectType> “=” <Value>

<create_sub_ref_expression> = <Variable>
                    |   <ObjectType> “==” STRING

<delete_expression> =   <Variable>
            |   <Variable> “.” <Relation>

<set_expression>    =   “(“ <set_expression> “)”
            |   <ObjectType> “=” <Value> “,” <set_expression >
            |   <ObjectType> “=” <Value>

<get_expression>    =   “(“ <get_expression> “)”
            |   <get_expression> AND <get_expression>
            |   <get_expression> OR <get_expression>
            |   NOT <get_expression>
```

```
            |   <ObjectType> “==” <Value>
            |   <ObjectType> “==” <Value> “,” <ObjectType> “==” <Value>
            |   <Variable> <Relation> <Variable>

<Relation>      =   “[“ <Function_Type> “,” <Mapping_Type> “=”
                        <Relation_Type> “,” STRING  “]”

<Mapping_Type>  =   $DR         // Direct Relation
            |   $IR         // Indirect Relation

<Relation_Type> =   $CONT           // Containment
            |   $REF            // Reference
            |   $CONN           // Connection

<Function_Type> =   $ONEONE         // One-to-one
            |   $ONEMANY        // One-to-many
            |   $MANYTOONE      // Many-to-one
            |   $MANYMANY       // Many-to-many

<ObjectType>        =   CT          // Component type
            |   CN          // Component name
            |   PN          // Property Name                           |   PV
        // Property Value

<Variable>      =   '$' STRING      // Return variable
            |   STRING

<Value>     =   INTEGER
            |   STRING
            |   “*”         // For wildcard specification
            |   <scalar_exp>

<scalar_exp>        =   <scalar_exp> “+” <scalar_exp>
            |   <scalar_exp> “-“ <scalar_exp>
            |   <scalar_exp> ”*” <scalar_exp>
            |   <scalar_exp> “/” <scalar_exp>
            |   “-“ <scalar_exp>
            |   “+” <scalar_exp>
            |   “(“ <scalar_exp> “)”
            |   <ObjectType>
            |   INTEGER
```